Systems programming

8 – Indirect communication

MEEC LEEC MEAer LEAer MEIC-A

João Nuno Silva

TÉCNICO LISBOA

Bibliography

- Distributed Systems Concepts and Design, George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair
- Chapter 6
- The Linux Programming Interface, Michael Kerrisk
- Chapter 52.1 .. 52.5

Point-to-point communication

- Pipes, Fifos, sockets
  - Participants need to exist at the same time
  - Participants need to know address of each other and identities
  - It is necessary to establish communication
-
- Not a good way to communicate with several participants
- Not a good way to implement complex communication

Indirect communication

- Space uncoupling
  - No need to know identity of receiver
  - Anonymous senders
  - Participants can be replaced, updated, replicated, or migrated
- Time uncoupling
  - independent lifetimes
  - requires persistence in the communication channel
- Communication through an intermediary
  - No direct coupling between the sender and the receivers

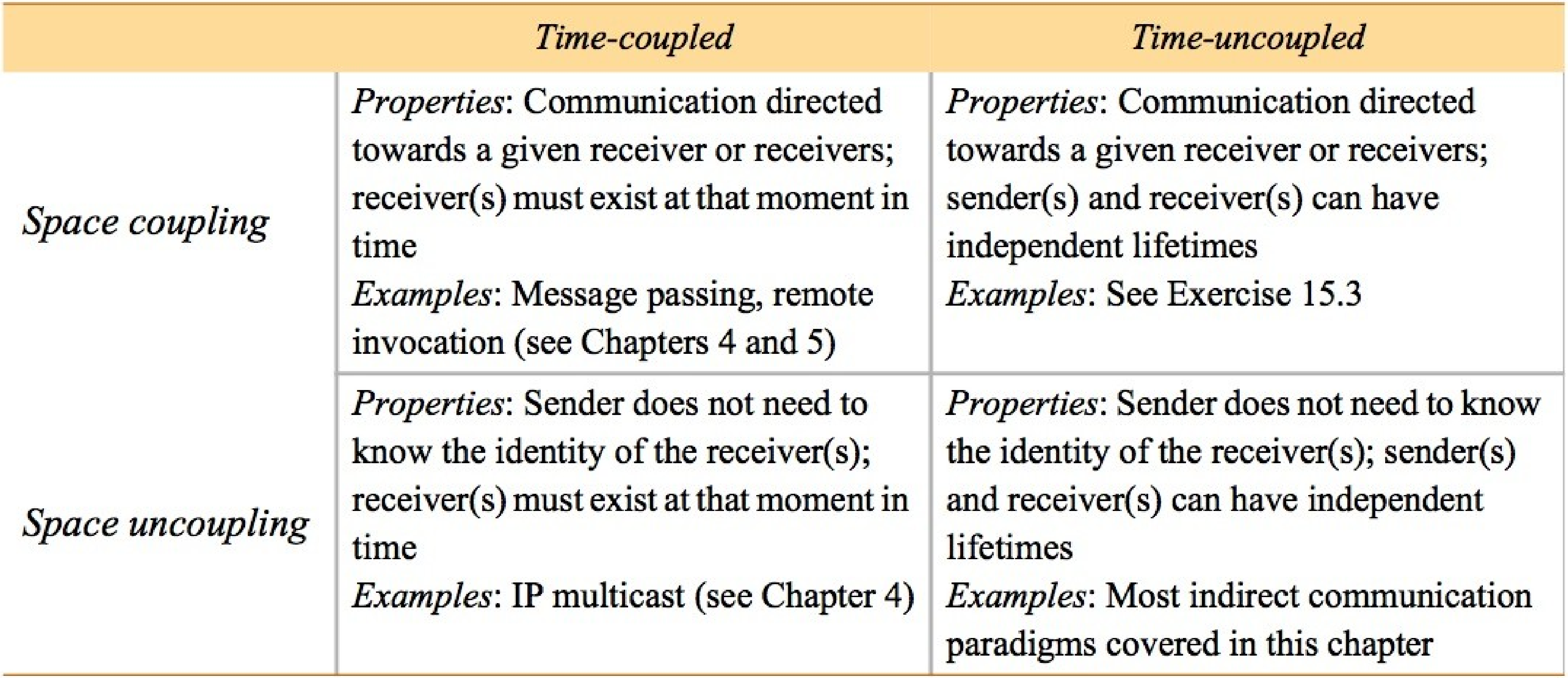
Indirect Communication

- Scenarios where users connect and disconnect very often
  - Mobile environments, messaging services, forums
- Event dissemination where receivers may be unknown and change often
  - RSS, events feeds in financial services
- Scenarios with very large number of participants
  - Google Ads system, Spotify
- Commonly used in cases when change is anticipated
  - need to provide dependable services

Indirect Communication

- Requires a middleware
- Performance overhead introduced by
  - adding a level of indirection
  - Implementation of reliable message delivery
  - ordering grantees
- More difficult to manage due to lack of direct coupling
- Difficult to achieve end-to-end properties
  - Real time behavior
  - Security

Indirect communication



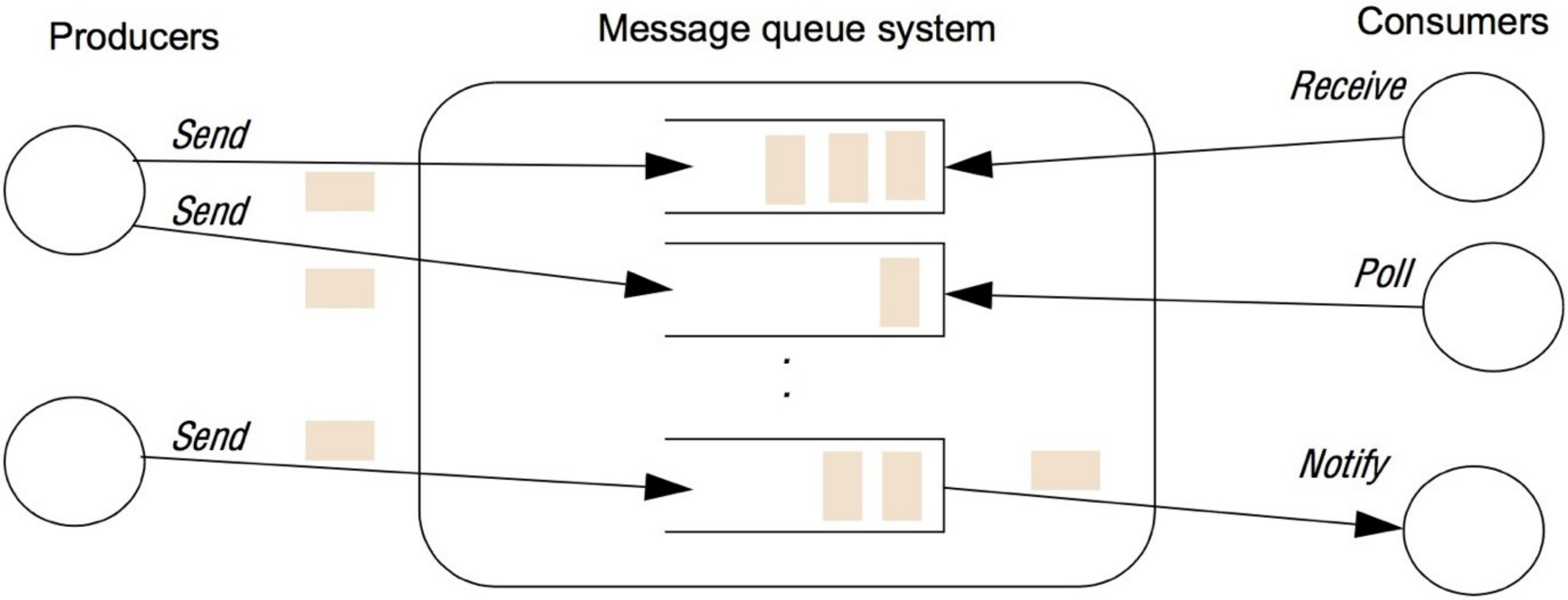
	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 15.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

Message Queues

Message Queues

- Intermediary between producers and consumers of data
  - Point-to-Point, not one-to-many
  - Supports time and space uncoupling
- Programming model
  - producer sends message to specific queue
  - consumers can
    - Block
    - Non-block (polling)
    - Notify

Message Queues



Message Queues

- Many processes can send to a queue
- Many can remove from it
- Queuing policy:
  - usually FIFO
  - Can also be priority-based
- Consumers can select based on metadata

Message Queues

- Messages are persistent
  - Stored until removed (on disk)
- Transaction support:
  - all-or-none operations
  - All message is completely delivered
- Automatic message transformation:
  - on arrival, message transforms data from one format to another (data heterogeneity)

POSIX Message Queues

- Array of bytes
- Priority / message selection
  - API
- Each message has an associated priority,
- Messages are always delivered to the receiving process highest priority first.
- Message priorities range
  - From 0 (low) to sysconf(_SC_MQ_PRIO_MAX) - 1 (high).
  - On Linux, sysconf(_SC_MQ_PRIO_MAX) returns 32768,
  - POSIX.1 requires a range from 0 to to 31
-

POSIX MQ – creation

- mq_open - open a message queue

  mqd_t mq_open(const char *name, int oflag, mode_t mode,
  struct mq_attr *attr);

- Name
  - identifier
- Oflags
  - **O_CREAT** | O_RDONLY | O_WRONLY | O_RDWR
- Mode
  - File access modes rwx / ugw 0666

POSIX MQ creation attributes

- 3rd argument of mq_open
  - Attributes of the message queus

```
struct mq_attr queue_attr;
    queue_attr.mq_maxmsg = 16;
    queue_attr.mq_msgsize = 128;
```

- Posix MQ are implemented as
  - Arrays os messages
  - Circular buffer

POSIX MQ – opening

- mq_open - open a message queue

  mqd_t mq_open(const char *name, int oflag)

  - Just 2 arguments
- Default settings
  - Name
    - identifier
  - Oflags -
    - O_RDONLY O_WRONLY O_RDWR

POSIX MQ lifecycle

- **mq_open**
  - Creates / open message queue
- Message queue is assigned to a file
  - In /dev/msgqueue/
  - File name is used by other processes
  - File object can be accessed

- mq_close
  - close a message queue descriptor
  - Process can no longer use queue
- mq_unlink
  - removes a message queue
  - Deletes the file

POSIX MQ - Message Structure

- Array of bytes
- Each message has an associated priority,
- Messages are always delivered to the receiving process highest priority first.
- Message priorities range
  - From 0 (low) to sysconf(_SC_MQ_PRIO_MAX) - 1 (high).
  - On Linux, sysconf(_SC_MQ_PRIO_MAX) returns 32768,
  - POSIX.1 requires a range from 0 to to 31

POSIX MQ - write

```
int mq_send(mqd_t mqdes,
            const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
```

- Writes a message to the queue
- Parameters
  - mqdes
    - queue id (returned from mq_open)
  - Message (byte array) + size
  - msg_priority

POSIX MQ - read

ssize_t mq_receive(mqd_t mqdes,

char *msg_ptr, size_t msg_len,

unsigned int *msg_prio);

- Reads a message from the queue
  - mqdes – queue id (returned from mq_open)
  - Message + buffer size
  - msg_prio - priority of received message)
- removes the oldest message with the highest priority
  - places it in the buffer pointed to by msg_ptr.

POSIX MQ - read

```
ssize_t mq_receive(mqd_t mqdes,
                   char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);
```

- If msg_priority not NULL
  - Used to store the priory of received message

Read/write

mq_receive

- If queue is empty
  - Receive blocks until
    - There is is a message
- **mq_timedreceive**
- Blocks until
  - Timeout
  - There is is a message

mq_send

- If queue is full
  - Send blocks until
    - queue has space for message

mq_timedsend

- Blocks until
  - Timeout
  - queue has space for message

POSIX MQ - limits

- Number of messages
- Size of each message
- Values limited by the OS
  - /proc/sys/fs/mqueue/
- Changed on:
  - /etc/security/limits.conf

- On the user program
  - queue_attr.mq_maxmsg = 16;
  - queue_attr.mq_msgsize = 128;

Group communication

Group communication

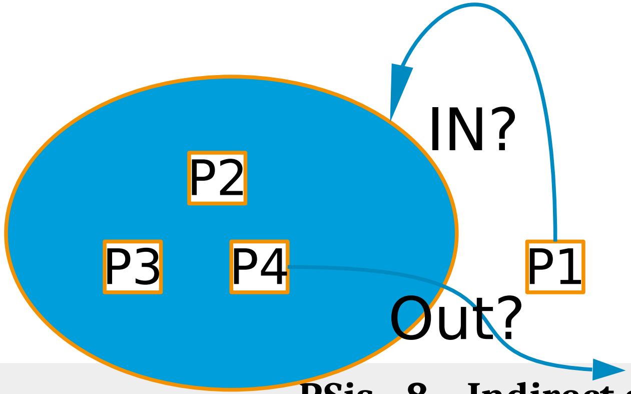
- Offers a service where
  - a message is sent to a group
  - then this message is delivered to all members of the group.
- Characteristics
  - Sender is not aware of the identities of the receivers
  - Represents an abstraction over multicast communication
- Added value in terms of
  - Managing group membership
  - Detecting failures and providing reliability and ordering guarantees

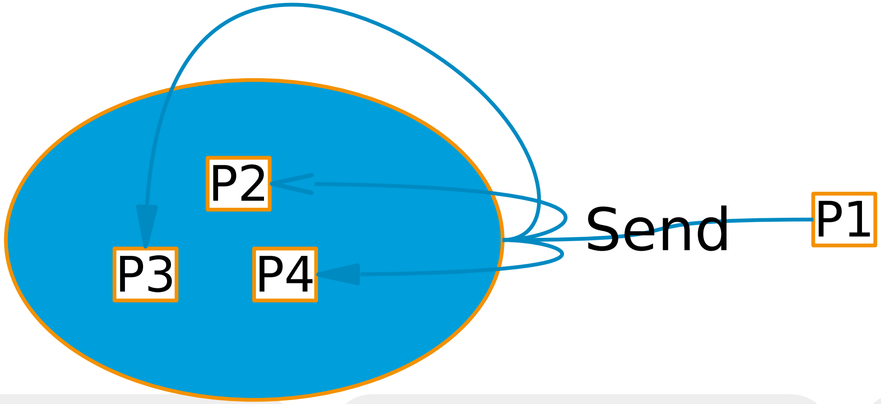
Group communication

- Reliable dissemination of information to potentially large numbers of clients,
  - financial industry that require accurate and up-to-date access to data
- Collaborative applications
  - where events must be disseminated to multiple users (e.g. multiuser games)
- Faulty tolerent message delivery
  - Consistent update of replicated data
  - Implementation of highly available (replicated) servers
- System monitoring and management
  - including for example load balancing strategies

Group Communication

- Central abstraction:
  - group & associated membership
- Processes join (explicitly) or leave (explicitly or by failure)
- Send single message to the group of N, not N unicast messages





Process groups

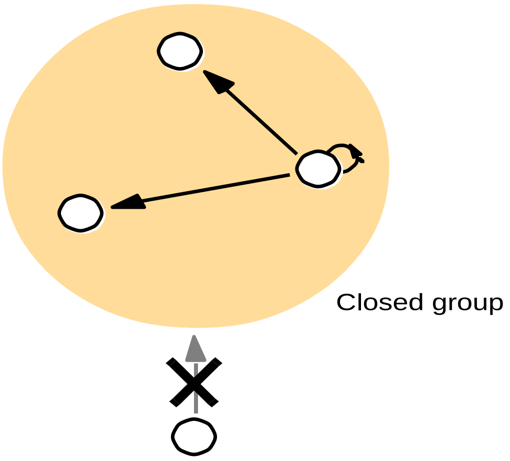
- Abstraction
  - resilient process
- Messages delivered to a process endpoint, no higher
- Messages
  - unstructured byte arrays
  - no marshaling etc
- Level of service
  - socket

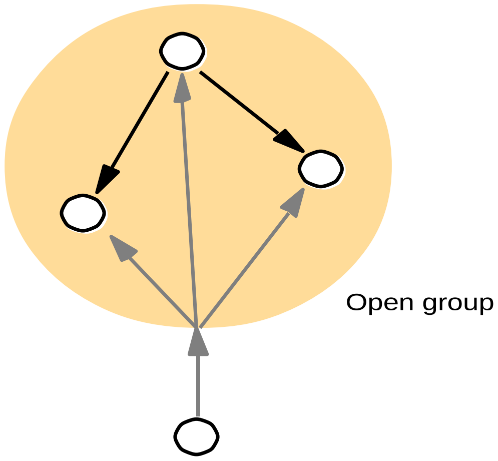
Object groups

- Higher level approach
- Collection of objects (same class!)
  - process same invocations
- Replication can be transparent to clients
- Invoke on single object (proxy)
- Requests sent by group communication
- Voting in proxy usually

Group membership

- Closed
  - Cooperating servers
  - Internal messages
- Open
  - Notification of services

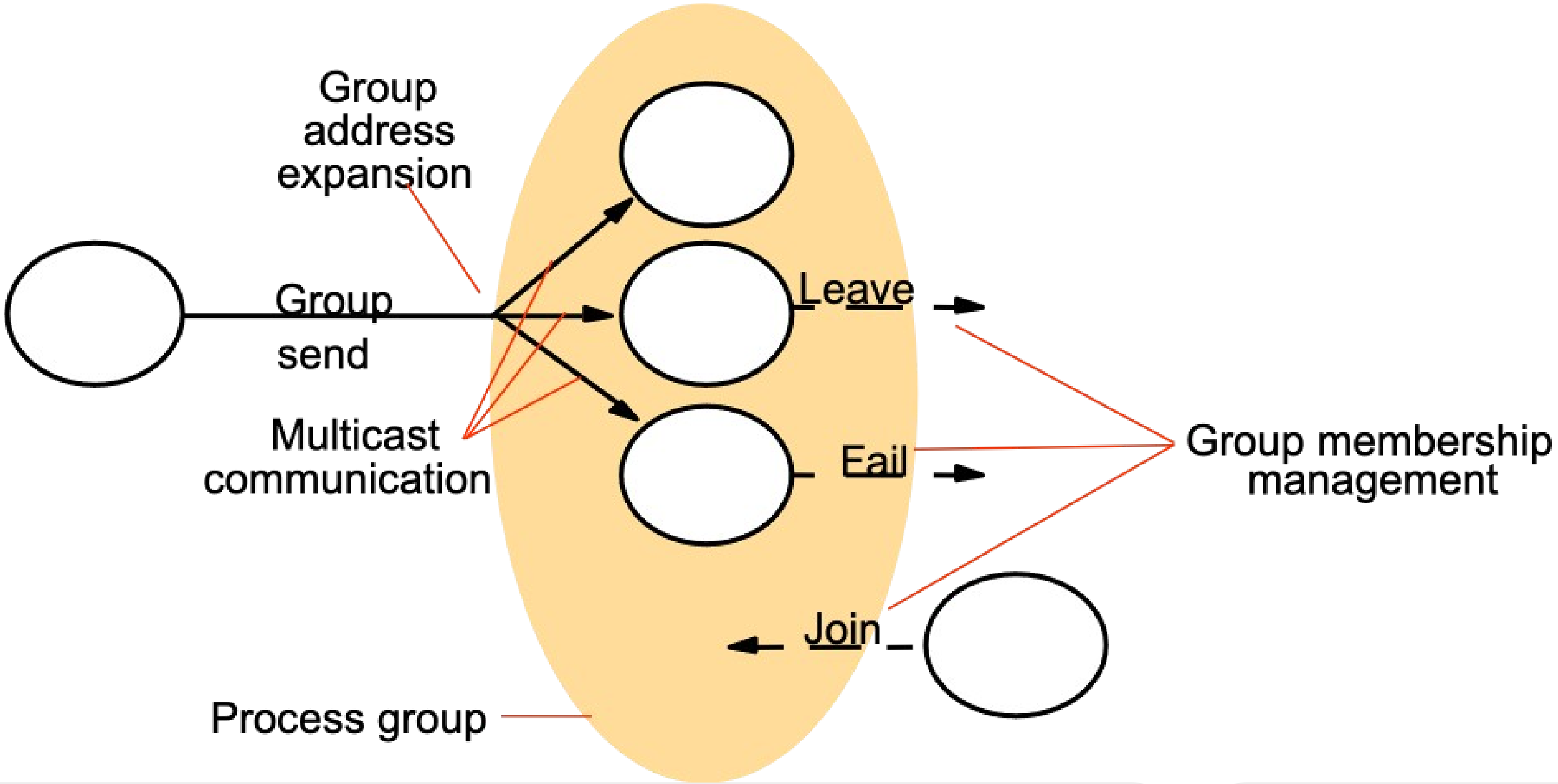




Group membership

- Interface for group membership changes
  - Operations to create and destroy process groups
  - add or withdraw a process to or from a group
- Failure detection
  - The service monitors the group members
    - Verify if they crash or become unreachable due to communication failure
- Notifying members of group membership changes
  - Notification of group's members when a processes are added, or removed
- Group address expansion
  - Replication of message set to group to all the members

Group membership



Communication Reliability

- Strong delivery reliability properties
  - Delivery integrity - message received same as sent, never delivered twice
  - Delivery validity - outgoing message eventually delivered
- Group communication reliability properties build on Unicast
  - Delivery integrity
    - Deliver message correctly at most once to group members
  - Delivery validity
    - message sent will be eventually delivered (if not all group members fail)
  - Agreement/consensus
    - Delivered to all or none of the group members

Message ordering

- FIFO ordering
  - first-in-first-out from a single sender to the group
- Causal ordering
  - preserves potential causality, happens before
- Total ordering
  - messages delivered in same order to all processes
- Perspective
  - Strong reliability and ordering is expensive: scale limited
  - More probabilistic approaches & weaker delivery possible

Publish-subscribe

Publish-subscribe

- Publish-subscribe
  - or distributed event systems
- Working fundamentals
  - Publishers publish structured events to event service
  - Subscribers express interest in particular events
  - Event service
    - matches published events to subscriptions
    - Delivers suitable events

Publish-subscribe

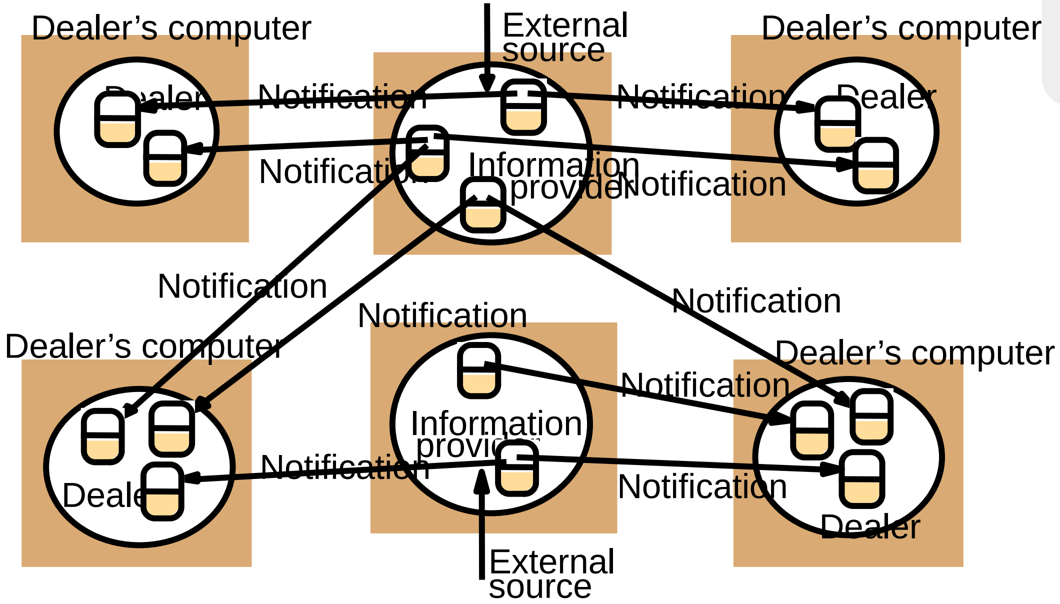
- Applications
  - Financial information systems
  - Live feeds of real-time data (including RSS)
  - Cooperative working
    - events of shared interest
  - Ubiquitous computing
    - location events, ... from infrastructure
  - Lots of monitoring applications

Stock trading system

- Let users see latest market prices of stock they care about
  - Info for a given stock arrives from multiple sources
  - Dealers only care about stocks they own (or might)
  - May only care to know above some threshold

Publish-subscribe

- Data provider
  - generate events (changes in stock value changes)
- Dealer process
  - creates subscription for each stock its user(s) express interest in



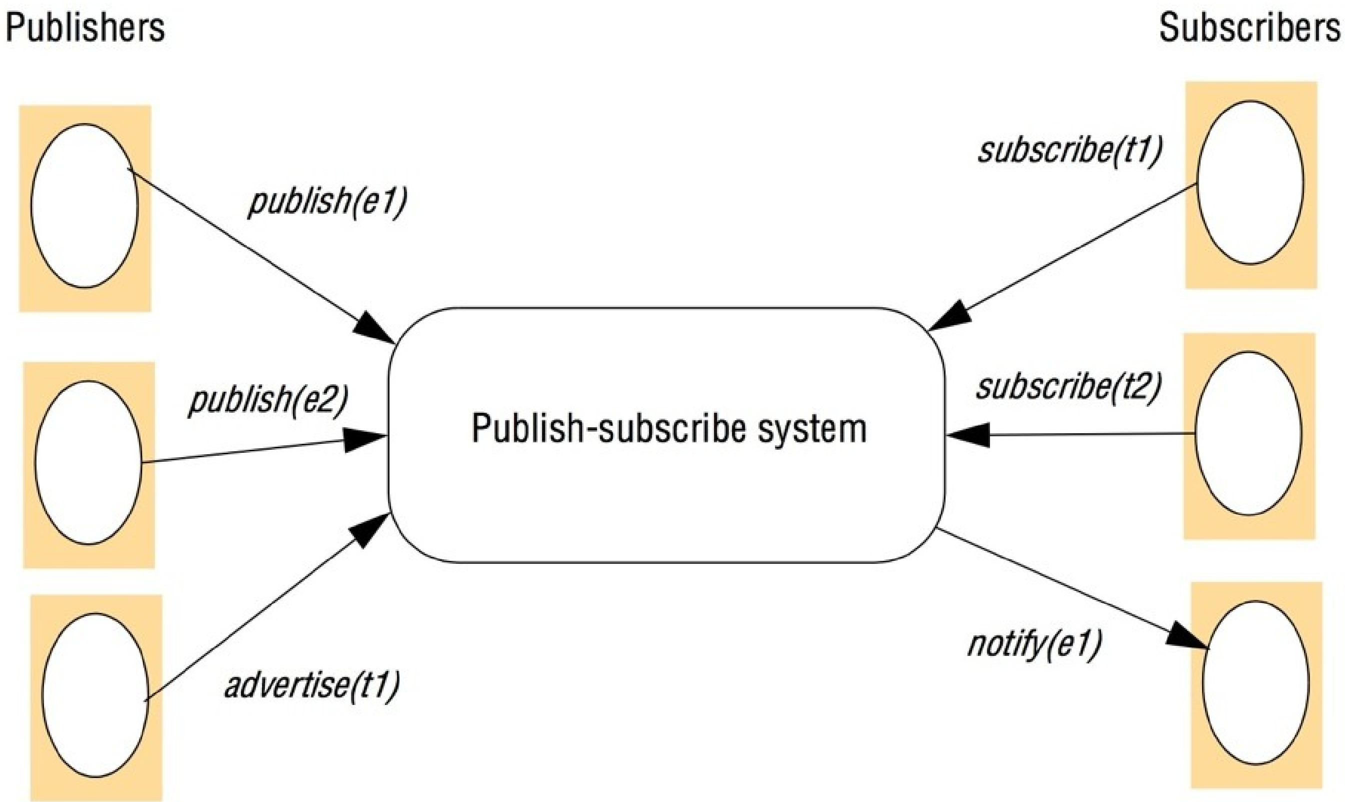
Publish-subscribe characteristics

- Heterogeneity
  - Able to glue together systems not designed to work together,
  - Have to come up with an external description of what can be subscribed to: simple flat, rich taxonomy, etc
- Asynchronism
  - Decoupling means you never have to block
- Possible delivery guarantees
  - All subscribers receive the same events (atomicity)
  - Events correctly delivered to subscribers at most once to subscribers (integrity)
  - message sent will be eventually delivered (validity)
  - Real-time

Programming model

- Publishers
  - Disseminate event **e** through **publish(e)**
  - Register/advertise via a filter (pattern over all events):
    - **f: advertise (f)**
    - Expressiveness of pattern is the subscription model
  - Can also remove the offer to publish: **unadvertise (f)**
- Subscribers
  - Subscribe events that follow a filter/pattern
  - Receive events that match filter
  - Cancel their subscription:

Programming model



Subscription model

- Channel-based
  - Publishers publish to named channels
  - Subscribers get ALL events from channel
  - Very simplistic, no filtering (all other models below do)
- Topic-based or subject-based
  - Each notification expressed in multiple fields
    - one is the event topic
  - Subscribers choose topics
  - Allows hierarchical topics can help
    - e.g., old USENET rec.sports.cricket

Subscription model

- Content-based
  - Generalization of topic based
  - Subscription is expression over range of fields (constraints on values)
  - Far more expressive than channel-based or topic-based
- Type-based
  - Use object-based approaches with object types
  - Subscriptions defined in terms of types of events
  - Matching in terms of types or subtypes of filter
  - Ranges from coarse grained (type names) to fine grained (attributes and methods of object)
  - Advantage: clean integration with object-based programming languages

Publish-subscribe - Main concerns

- Deliver events efficiently to all subscribers that have filters that match the events

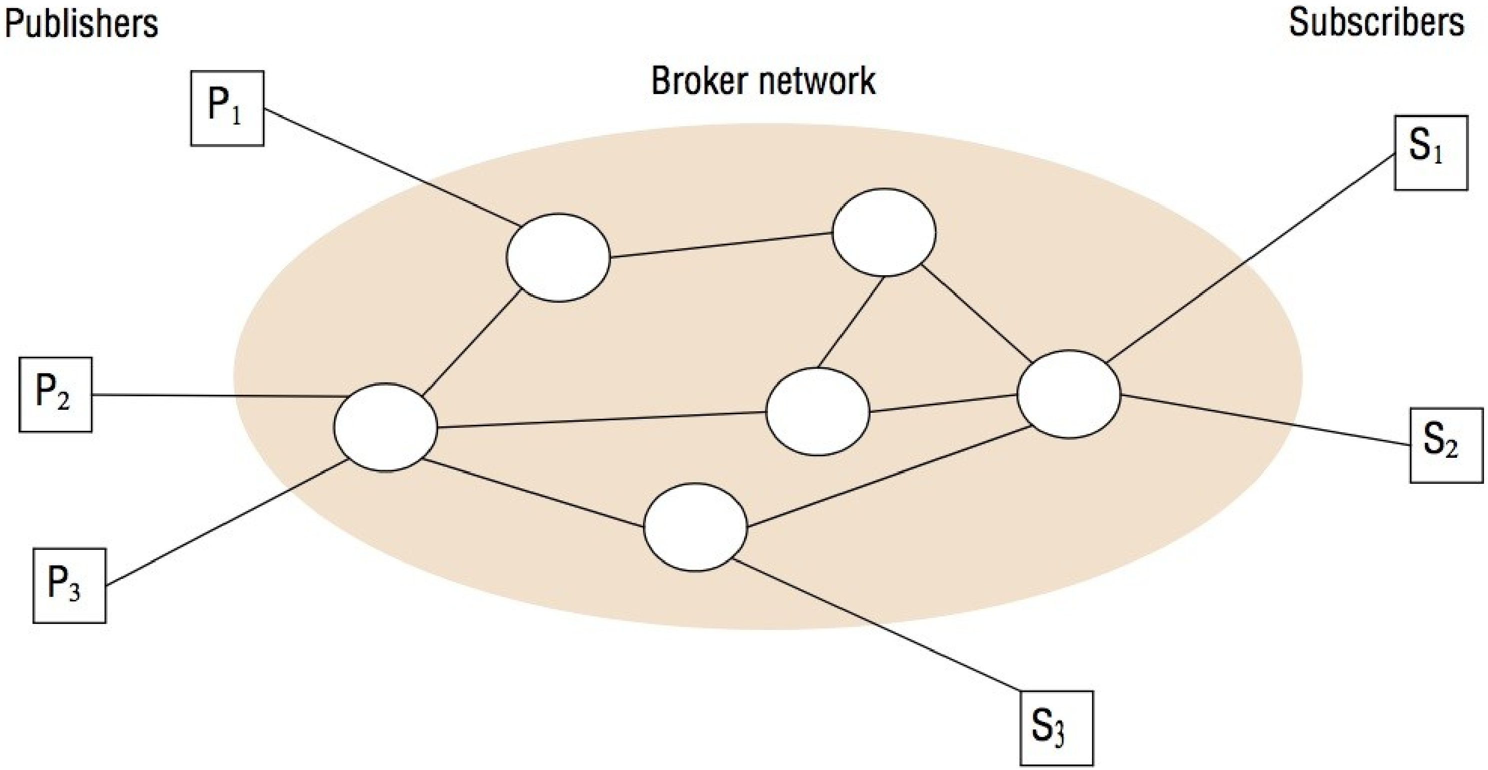
- Security
- Scalability
- Failure handling
- Quality of Service (QoS)

- Tradeoffs:
  - Latency vs Reliability
  - Ease in implementation vs Expressiveness of events types

Architecture

- Centralized schemes simple
  - Implementing channel-based or topic-based simple
  - Map channels/topics onto groups
  - Use the group's multicast (possibly reliable, ordered, ..)
  - Implementation of content/type/ more complicated
- Distributed
  - most implementations are network of brokers
- Some implementations are peer-to-peer (P2P)
  - All publisher and subscriber nodes act as the pub-sub broker
-

Publish-subscribe Distributed


Publishers
Subscribers
Broker network
P_1
P_2
P_3
S_1
S_2
S_3